

# Apprentissage pour l'image

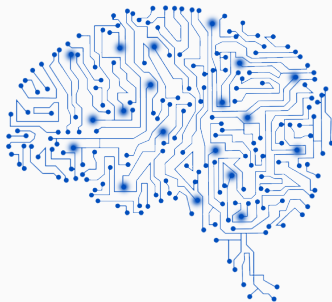
## Machine learning for image processing

### Course II – Introduction to Artificial Neural Networks: Backpropagation

---

Emile Pierret

Jeudi 16 mars 2023



At the end of the course :

- Know what is a CNN (Convolutional Neural Network)
- Implement the training of a CNN for classification with Pytorch

For this session :

- Backpropagation to compute gradients in Neural Networks

## A simple example

Let consider  $f : (x, y) \in \mathbb{R}^2 \mapsto \log(xy)$ . How to differentiate  $f$  step by step with respect to  $x$  and  $y$  ?

## A simple example

Let consider  $f : (x, y) \in \mathbb{R}^2 \mapsto \log(xy)$ . How to differentiate  $f$  step by step with respect to  $x$  and  $y$  ? We can split  $f$  such that :

❶  $v = xy$

## A simple example

Let consider  $f : (x, y) \in \mathbb{R}^2 \mapsto \log(xy)$ . How to differentiate  $f$  step by step with respect to  $x$  and  $y$  ? We can split  $f$  such that :

❶  $v = xy$

❷  $w = \log(v) = \log(xy)$

## A simple example

Let consider  $f : (x, y) \in \mathbb{R}^2 \mapsto \log(xy)$ . How to differentiate  $f$  step by step with respect to  $x$  and  $y$  ? We can split  $f$  such that :

❶  $v = xy$

❷  $w = \log(v) = \log(xy)$

❸  $z = w$

## A simple example

Let consider  $f : (x, y) \in \mathbb{R}^2 \mapsto \log(xy)$ . How to differentiate  $f$  step by step with respect to  $x$  and  $y$  ? We can split  $f$  such that :

❶  $v = xy$

❷  $w = \log(v) = \log(xy)$

❸  $z = w$

As a reminder, for  $g, h : \mathbb{R} \rightarrow \mathbb{R}$ ,

$$(g \circ f)'(x) = g'(f(x))f'(x)$$

## A simple example

Let consider  $f : (x, y) \in \mathbb{R}^2 \mapsto \log(xy)$ . How to differentiate  $f$  step by step with respect to  $x$  and  $y$  ? We can split  $f$  such that :

❶  $v = xy$

❷  $w = \log(v) = \log(xy)$

❸  $z = w$

As a reminder, for  $g, h : \mathbb{R} \rightarrow \mathbb{R}$ ,

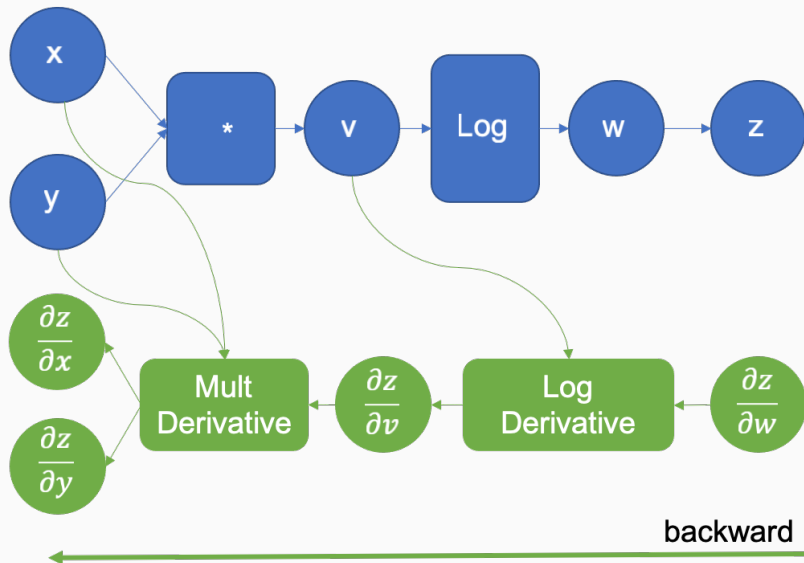
$$(g \circ f)'(x) = g'(f(x))f'(x)$$

Consequently,

$$\begin{aligned}\frac{\partial z}{\partial x} &= \frac{\partial z}{\partial w} \frac{\partial w}{\partial x} \\ &= \frac{\partial z}{\partial w} \frac{\partial w}{\partial v} \frac{\partial v}{\partial x} \\ &= 1 \times \frac{1}{v} \times y \\ &= \frac{1}{x}\end{aligned}$$

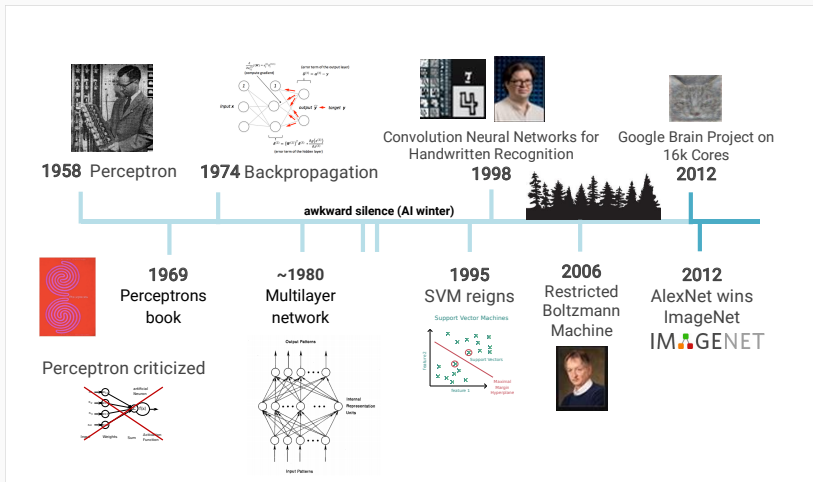


forward



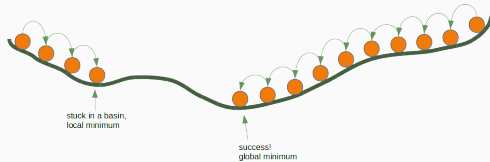
backward

## Timeline of (deep) learning

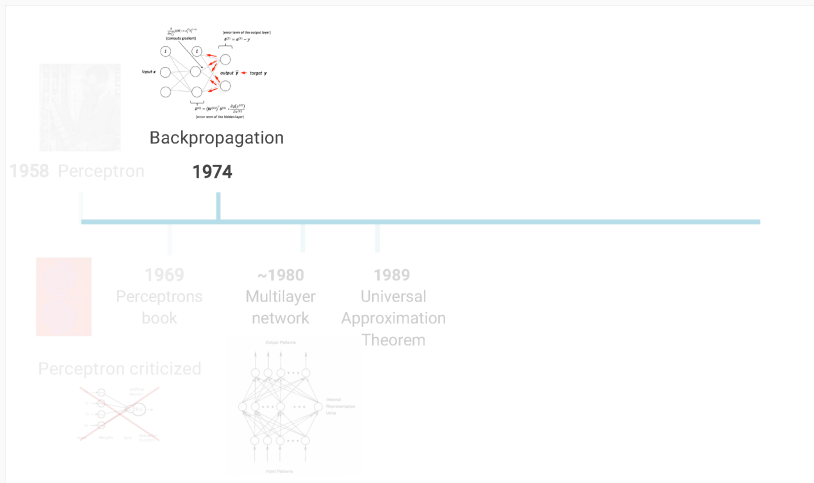


# Backpropagation

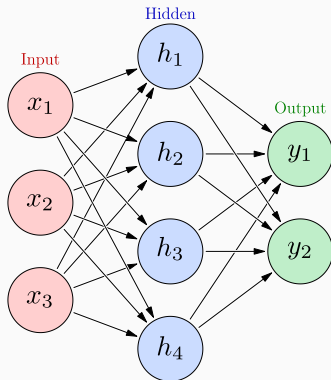
---



## Learning with backpropagation



## Artificial neural network / Multilayer perceptron / NeuralNet



$$h_1 = g_1 (w_{11}^1 x_1 + w_{12}^1 x_2 + w_{13}^1 x_3 + b_1^1)$$

$$h_2 = g_1 (w_{21}^1 x_1 + w_{22}^1 x_2 + w_{23}^1 x_3 + b_2^1)$$

$$h_3 = g_1 (w_{31}^1 x_1 + w_{32}^1 x_2 + w_{33}^1 x_3 + b_3^1)$$

$$h_4 = g_1 (w_{41}^1 x_1 + w_{42}^1 x_2 + w_{43}^1 x_3 + b_4^1)$$

---

$$y_1 = g_2 (w_{11}^2 h_1 + w_{12}^2 h_2 + w_{13}^2 h_3 + w_{14}^2 h_4 + b_1^2)$$

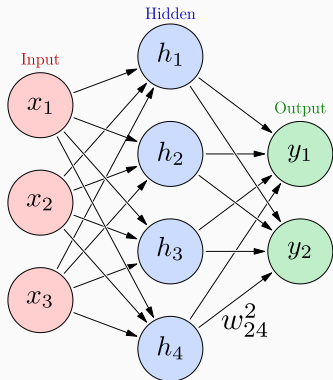
$$y_2 = g_2 (w_{21}^2 h_1 + w_{22}^2 h_2 + w_{23}^2 h_3 + w_{24}^2 h_4 + b_2^2)$$

---

$w_{ij}^k$  synaptic weight between previous node  $j$  and next node  $i$  at layer  $k$ .

$g_k$  are any activation function applied to each coefficient of its input vector.

## Artificial neural network / Multilayer perceptron / NeuralNet



$$h_1 = g_1 (w_{11}^1 x_1 + w_{12}^1 x_2 + w_{13}^1 x_3 + b_1^1)$$

$$h_2 = g_1 (w_{21}^1 x_1 + w_{22}^1 x_2 + w_{23}^1 x_3 + b_2^1)$$

$$h_3 = g_1 (w_{31}^1 x_1 + w_{32}^1 x_2 + w_{33}^1 x_3 + b_3^1)$$

$$h_4 = g_1 (w_{41}^1 x_1 + w_{42}^1 x_2 + w_{43}^1 x_3 + b_4^1)$$

---

$$y_1 = g_2 (w_{11}^2 h_1 + w_{12}^2 h_2 + w_{13}^2 h_3 + w_{14}^2 h_4 + b_1^2)$$

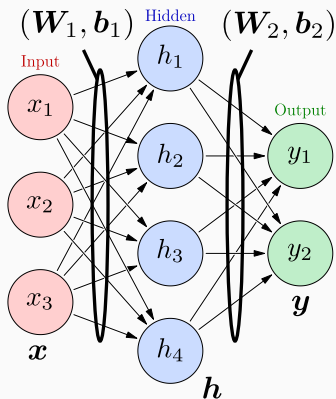
$$y_2 = g_2 (w_{21}^2 h_1 + w_{22}^2 h_2 + w_{23}^2 h_3 + w_{24}^2 h_4 + b_2^2)$$

---

$w_{ij}^k$  synaptic weight between previous node  $j$  and next node  $i$  at layer  $k$ .

$g_k$  are any activation function applied to each coefficient of its input vector.

## Artificial neural network / Multilayer perceptron / NeuralNet



$$h_1 = g_1 (w_{11}^1 x_1 + w_{12}^1 x_2 + w_{13}^1 x_3 + b_1^1)$$

$$h_2 = g_1 (w_{21}^1 x_1 + w_{22}^1 x_2 + w_{23}^1 x_3 + b_2^1)$$

$$h_3 = g_1 (w_{31}^1 x_1 + w_{32}^1 x_2 + w_{33}^1 x_3 + b_3^1)$$

$$h_4 = g_1 (w_{41}^1 x_1 + w_{42}^1 x_2 + w_{43}^1 x_3 + b_4^1)$$

---

$$\mathbf{h} = g_1 (\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1)$$

$$y_1 = g_2 (w_{11}^2 h_1 + w_{12}^2 h_2 + w_{13}^2 h_3 + w_{14}^2 h_4 + b_1^2)$$

$$y_2 = g_2 (w_{21}^2 h_1 + w_{22}^2 h_2 + w_{23}^2 h_3 + w_{24}^2 h_4 + b_2^2)$$

---

$$\mathbf{y} = g_2 (\mathbf{W}_2 \mathbf{h} + \mathbf{b}_2)$$

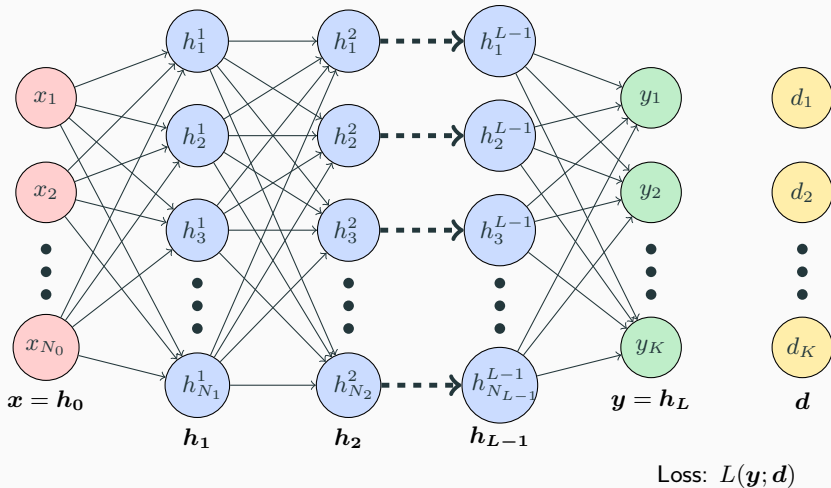
$w_{ij}^k$  synaptic weight between previous node  $j$  and next node  $i$  at layer  $k$ .

$g_k$  are any activation function applied to each coefficient of its input vector.

The matrices  $\mathbf{W}_k$  and biases  $\mathbf{b}_k$  are learned from labeled training data.

# Feedforward Artificial Neural Network

Recall the feedforward structure



Input Layer

Hidden Layers

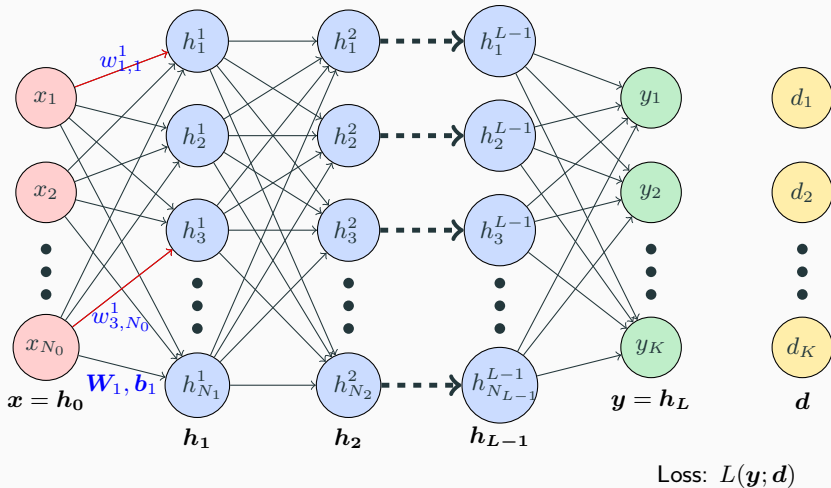
Output Layer

Label



# Feedforward Artificial Neural Network

Recall the feedforward structure



Input Layer

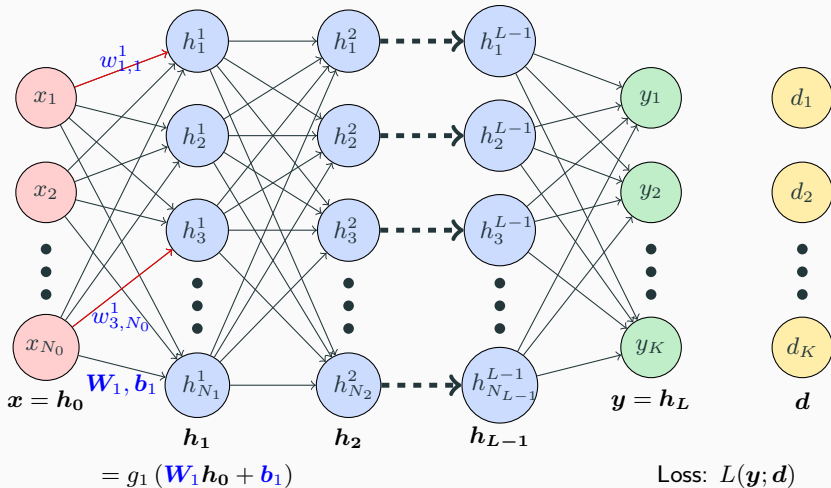
Hidden Layers

Output Layer

Label

# Feedforward Artificial Neural Network

Recall the feedforward structure



Input Layer

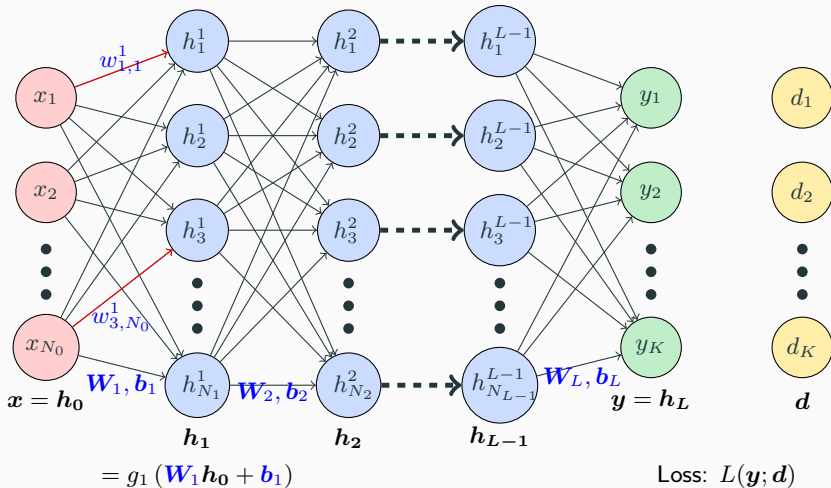
Hidden Layers

Output Layer

Label

# Feedforward Artificial Neural Network

Recall the feedforward structure



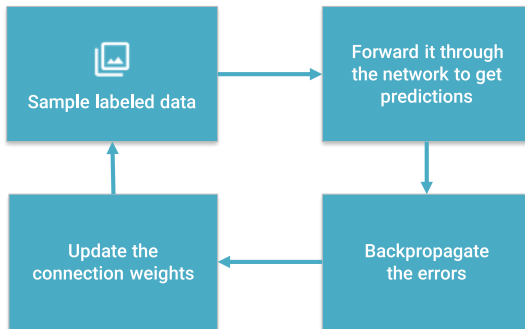
Input Layer

Hidden Layers

Output Layer

Label

## Training process



Learns by generating an error signal that measures the difference between the predictions of the network and the desired values and then **using this error signal to change the weights** (or parameters) so that predictions get more accurate.

- The parameters of the neural network are

$$\mathbf{W} = (\mathbf{W}_1, \mathbf{b}_1, \mathbf{W}_2, \mathbf{b}_2, \dots, \mathbf{W}_L, \mathbf{b}_L)$$

- Training the network = minimizing the training loss  $E(\mathbf{W})$

**Objective:**  $\min_{\mathbf{W}} E(\mathbf{W})$  where  $E(\mathbf{W}) = \sum_{(\mathbf{x}^i, \mathbf{d}^i) \in \mathcal{T}} L(\mathbf{y}^i; \mathbf{d}^i)$

$$\Rightarrow \nabla E(\mathbf{W}) = \left( \frac{\partial E(\mathbf{W})}{\partial \mathbf{W}_1} \quad \frac{\partial E(\mathbf{W})}{\partial \mathbf{b}_1} \quad \dots \quad \frac{\partial E(\mathbf{W})}{\partial \mathbf{W}_L} \quad \frac{\partial E(\mathbf{W})}{\partial \mathbf{b}_L} \right)^T = 0$$

- **Solution:** no closed-form solutions  $\Rightarrow$  use (stochastic) gradient descent.
- $\frac{\partial E(\mathbf{W})}{\partial \mathbf{W}_k}$  not really rigorous, we will use the notation

$$\nabla_{\mathbf{W}_k} E(\mathbf{W}) \quad \text{and} \quad \nabla_{\mathbf{b}_k} E(\mathbf{W}).$$

## Minimizing training loss

For multilayer neural networks  $\mathbf{W} \mapsto E(\mathbf{W})$  is non-convex

⇒ No guarantee of convergence.

Even if convergence occurs, the solution depends on the initialization and the step size/learning rate  $\gamma$ .

Nevertheless, really good minima or saddle points are reached in practice by

$$\mathbf{W}^{t+1} \leftarrow \mathbf{W}^t - \gamma \nabla E(\mathbf{W}^t), \quad \gamma > 0$$

Gradient descent can be expressed coordinate by coordinate as:

$$w_{i,j}^{k,t+1} \leftarrow w_{i,j}^{k,t} - \gamma \frac{\partial E(\mathbf{W}^t)}{\partial w_{i,j}^k}$$

for all weights  $w_{i,j}^k$  linking a node  $j$  to a node  $i$  in the next layer  $k$ .

⇒ The algorithm to compute  $\frac{\partial E(\mathbf{W})}{\partial w_{i,j}^k}$  for ANNs is called **backpropagation**.

- In practice we only use **stochastic gradient descent** with batch of training set.
- The complete loss is :

$$E(W) = \sum_{(\mathbf{x}^i, \mathbf{d}^i) \in \mathcal{T}} L(\mathbf{y}^i; \mathbf{d}^i)$$

- For some random small subset (e.g. batch)  $\mathcal{S} \subset \mathcal{T}$ , consider

$$E(\mathbf{W}; \mathcal{S}) = \sum_{(\mathbf{x}^i, \mathbf{d}^i) \in \mathcal{S}} L(\mathbf{y}^i; \mathbf{d}^i)$$

- Our **goal** is to compute the gradient

$$\nabla_{\mathbf{W}_k} E(\mathbf{W}; \mathcal{S}) = \sum_{(\mathbf{x}^i, \mathbf{d}^i) \in \mathcal{S}} \nabla_{\mathbf{W}_k} L(\mathbf{y}^i; \mathbf{d}^i).$$

- In practice we only use **stochastic gradient descent** with batch of training set.
- The complete loss is :

$$E(W) = \sum_{(\mathbf{x}^i, \mathbf{d}^i) \in \mathcal{T}} L(\mathbf{y}^i; \mathbf{d}^i)$$

- For some random small subset (e.g. batch)  $\mathcal{S} \subset \mathcal{T}$ , consider

$$E(\mathbf{W}; \mathcal{S}) = \sum_{(\mathbf{x}^i, \mathbf{d}^i) \in \mathcal{S}} L(\mathbf{y}^i; \mathbf{d}^i)$$

- Our **goal** is to compute the gradient

$$\nabla_{\mathbf{W}_k} E(\mathbf{W}; \mathcal{S}) = \sum_{(\mathbf{x}^i, \mathbf{d}^i) \in \mathcal{S}} \nabla_{\mathbf{W}_k} L(\mathbf{y}^i; \mathbf{d}^i).$$

- Why is this relevant to minimize  $E(\mathbf{W}) = E(\mathbf{W}; \mathcal{T})$  ?



- **Stochastic gradient descent:** For some random small subset (e.g. batch)  $\mathcal{S} \subset \mathcal{T}$ , our **goal** is to compute the noisy gradient

$$\nabla_{\mathbf{W}_k} E(\mathbf{W}; \mathcal{S}) = \sum_{(\mathbf{x}^i, \mathbf{d}^i) \in \mathcal{S}} \nabla_{\mathbf{W}_k} L(\mathbf{y}^i; \mathbf{d}^i).$$

- **Unbiased approximation:** As soon as  $\mathcal{S}$  spans uniformly the whole training set  $\mathcal{T}$ ,

$$\begin{aligned} \mathbb{E}_{\mathcal{S}} (\nabla_{\mathbf{W}_k} E(\mathbf{W}; \mathcal{S})) &= \mathbb{E}_{\mathcal{S}} \left( \sum_{(\mathbf{x}^i, \mathbf{d}^i) \in \mathcal{S}} \nabla_{\mathbf{W}_k} L(\mathbf{y}^i; \mathbf{d}^i) \right) \\ &= \mathbb{E}_{\mathcal{S}} \left( \sum_{(\mathbf{x}^i, \mathbf{d}^i) \in \mathcal{T}} \mathbf{1}_{(\mathbf{x}^i, \mathbf{d}^i) \in \mathcal{S}} \nabla_{\mathbf{W}_k} L(\mathbf{y}^i; \mathbf{d}^i) \right) \\ &= \frac{|\mathcal{S}|}{|\mathcal{T}|} \sum_{(\mathbf{x}^i, \mathbf{d}^i) \in \mathcal{T}} \nabla_{\mathbf{W}_k} L(\mathbf{y}^i; \mathbf{d}^i) = \frac{|\mathcal{S}|}{|\mathcal{T}|} \nabla_{\mathbf{W}_k} E(\mathbf{W}). \end{aligned}$$

- **Conclusion:** In expectation the noisy gradient is equal to the gradient using the whole training dataset (unbiased estimator).

**Loss functions:** Classical loss functions are:

**For regression:**  $\mathbf{d}^i \in \mathbb{R}^K$

- Square error

$$E(\mathbf{W}) = \sum_{(\mathbf{x}^i, \mathbf{d}^i) \in \mathcal{T}} \frac{1}{2} \|\mathbf{y}^i - \mathbf{d}^i\|_2^2 = \sum_{(\mathbf{x}^i, \mathbf{d}^i) \in \mathcal{T}} \frac{1}{2} \sum_k (y_k^i - d_k^i)^2$$

**For multi-class classification:**  $d^i \in \{1, \dots, K\}$ , coded by  $\mathbf{d}^i \in \{0, 1\}^K$ ,

- Cross-entropy with softmax as the last layer

$$E(\mathbf{W}) = - \sum_{(\mathbf{x}^i, \mathbf{d}^i)} \sum_{k=1}^K d_k^i \log y_k^i \quad \text{with} \quad \mathbf{y}^i = f(\mathbf{x}^i; \mathbf{W}) = \text{softmax}(\mathbf{a}^i) \in (0, 1)^K.$$

- Cross-entropy with softmax included in loss (PyTorch convention):

$\mathbf{y}^i = \mathbf{a}^i$  is the output of the last linear layer:

$$E(\mathbf{W}) = - \sum_{(\mathbf{x}^i, \mathbf{d}^i)} \left[ a_{d^i} - \log \left( \sum_{k=1}^K \exp(a_k) \right) \right] \quad \text{with } d^i \text{ the class of } \mathbf{x}^i.$$

- The loss functions are of the form

$$E(\mathbf{W}) = \sum_{(\mathbf{x}^i, \mathbf{d}^i)} L(\mathbf{y}^i; \mathbf{d}^i)$$

- By linearity,

$$\nabla E(\mathbf{W}) = \sum_{(\mathbf{x}^i, \mathbf{d}^i)} \nabla L(\mathbf{y}^i; \mathbf{d}^i)$$

- There the neural net output  $\mathbf{y}^i = f(\mathbf{x}^i; \mathbf{W})$  is a function of the input data  $\mathbf{x}^i$  and the neural weights  $\mathbf{W}$ .
- We know the gradient of  $L(\mathbf{y}^i; \mathbf{d}^i)$  with respect to the variable  $\mathbf{y}$ 
  - Regression/Square error:

$$L(\mathbf{y}; \mathbf{d}) = \frac{1}{2} \|\mathbf{y} - \mathbf{d}\|_2^2 \quad \Rightarrow \quad \nabla_{\mathbf{y}} L(\mathbf{y}; \mathbf{d}) = \mathbf{y} - \mathbf{d}$$

- Multi-class classification/cross-entropy:

$$L(\mathbf{y}; \mathbf{d}) = -y_d + \log \left( \sum_{k=1}^K \exp(y_k) \right) \quad \Rightarrow \quad \nabla_{\mathbf{y}} L(\mathbf{y}; \mathbf{d}) = \text{softmax}(\mathbf{y}) - \mathbf{d}.$$

- The loss functions are of the form

$$E(\mathbf{W}) = \sum_{(\mathbf{x}^i, \mathbf{d}^i)} L(\mathbf{y}^i; \mathbf{d}^i)$$

- By linearity,

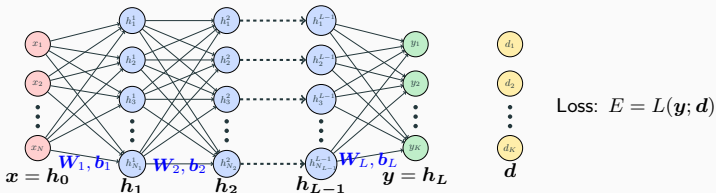
$$\nabla E(\mathbf{W}) = \sum_{(\mathbf{x}^i, \mathbf{d}^i)} \nabla L(\mathbf{y}^i; \mathbf{d}^i)$$

- There the neural net output  $\mathbf{y}^i = f(\mathbf{x}^i; \mathbf{W})$  is a function of the input data  $\mathbf{x}^i$  and the neural weights  $\mathbf{W}$ .
- We know the gradient of  $L(\mathbf{y}^i; \mathbf{d}^i)$  with respect to the variable  $\mathbf{y}$
- We still need to compute

$$\nabla_{\mathbf{W}_k} L(\mathbf{y}; \mathbf{d}) \quad \text{and} \quad \nabla_{\mathbf{b}_k} L(\mathbf{y}; \mathbf{d}) \quad \text{for } k = 0, \dots, L.$$

- For simplicity above we will use the notation  $E = L(\mathbf{y}; \mathbf{d})$ , that is considering only one point.

# ANN – Backpropagation



## Forward pass

Initialization:

$$h_0 = x$$

**for** layer  $k = 1$  **to**  $L$  **do**

Linear unit:

$$a_k = W_k h_{k-1} + b_k$$

Componentwise non-linear activation:

$$h_k = g_k(a_k)$$

**end**

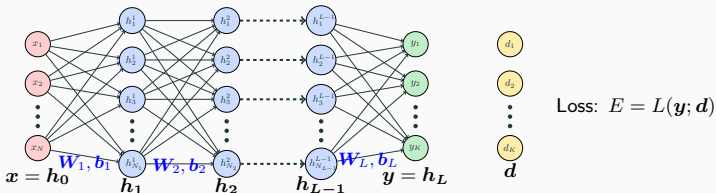
Output layer:

$$y = h_L$$

Compute loss:

$$E = L(y; d)$$

# ANN – Backpropagation



## Forward pass

Initialization:

$$\mathbf{h}_0 = \mathbf{x}$$

**for** layer  $k = 1$  **to**  $L$  **do**

Linear unit:

$$\mathbf{a}_k = \mathbf{W}_k \mathbf{h}_{k-1} + \mathbf{b}_k$$

Componentwise non-linear activation:

$$\mathbf{h}_k = g_k(\mathbf{a}_k)$$

**end**

Output layer:

$$\mathbf{y} = \mathbf{h}_L$$

Compute loss:

$$E = L(\mathbf{y}; \mathbf{d})$$

## Backward pass

**Goal:** Compute the gradient with respect to all parameters

$$\frac{\partial E}{\partial w_{i,j}^k} = ? \quad \frac{\partial E}{\partial b_i^k} = ?$$

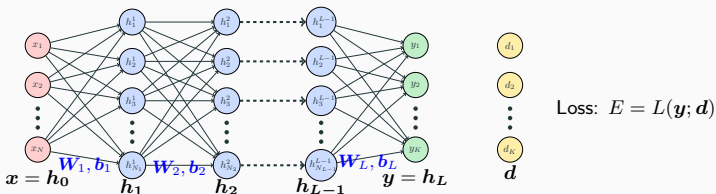
for all

$$k \in \{1, \dots, L\},$$

$$i \in \{1, \dots, N_k\},$$

$$j \in \{1, \dots, N_{k-1}\}.$$

# ANN – Backpropagation

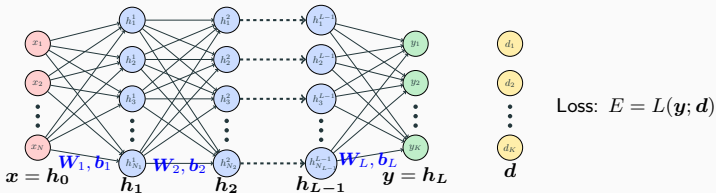


## Going backward

- We know how to compute the loss function and its gradient:

$$\nabla_{h_L} E = \nabla L(y; d)$$

# ANN – Backpropagation



Gradient with respect to last linear unit output  $a_L$

$$h_L = g_L(a_L)$$

That is for all  $i \in \{1, \dots, N_L\}$ ,  $h_i^L = g_L(a_i^L)$ . By the chain rule,

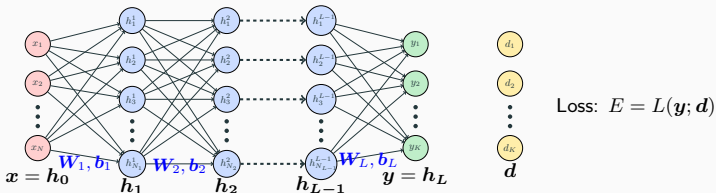
$$\frac{\partial E}{\partial a_i^L} = \frac{\partial E}{\partial h_i^L} \frac{\partial h_i^L}{\partial a_i^L} = [\nabla_{h_L} E]_i g'_L(a_i^L)$$

**Vector formula:**  $\nabla_{a_L} E = \nabla_{h_L} E \odot g'_L(a_L)$

where  $\odot$  is the componentwise product between vectors, ie Hadamard product.



# ANN – Backpropagation



Gradient with respect to bias of last linear unit  $b_L$

$$\mathbf{a}_L = \mathbf{W}_L \mathbf{h}_{L-1} + \mathbf{b}_L$$

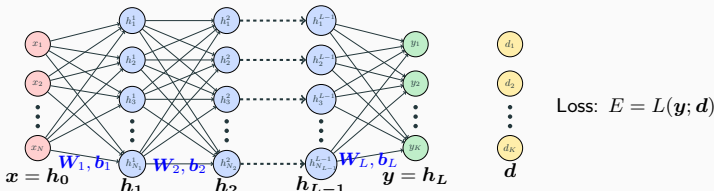
That is for all  $i \in \{1, \dots, N_L\}$ ,  $a_i^L = \sum_{j=1}^{N_{L-1}} w_{i,j}^L h_j^{L-1} + b_i^L$ .

By the chain rule, for all  $i \in \{1, \dots, N_L\}$ ,

$$\frac{\partial E}{\partial b_i^L} = \frac{\partial E}{\partial a_i^L} \underbrace{\frac{\partial a_i^L}{\partial b_i^L}}_{=1} = \frac{\partial E}{\partial a_i^L} = [\nabla_{\mathbf{a}_L} E]_i$$

Vector formula:  $\nabla_{\mathbf{b}_L} E = \nabla_{\mathbf{a}_L} E$

# ANN – Backpropagation



Gradient with respect to weights of last linear unit  $W_L$

$$a_L = W_L h_{L-1} + b_L$$

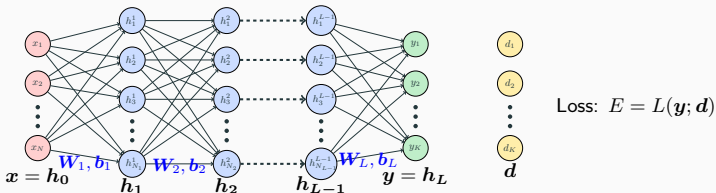
That is for all  $i \in \{1, \dots, N_L\}$ ,  $a_i^L = \sum_{j=1}^{N_{L-1}} w_{i,j}^L h_j^{L-1} + b_i^L$ .

By the chain rule, for all  $i \in \{1, \dots, N_L\}$  and  $j \in \{1, \dots, N_{L-1}\}$

$$\frac{\partial E}{\partial w_{i,j}^L} = \frac{\partial E}{\partial a_i^L} \underbrace{\frac{\partial a_i^L}{\partial w_{i,j}^L}}_{=h_j^{L-1}} = \frac{\partial E}{\partial a_i^L} h_j^{L-1} = [\nabla_{a_L} E]_i [h_{L-1}]_j$$

Matrix formula:  $\nabla_{W_L} E = \nabla_{a_L} E h_{L-1}^T$

# ANN – Backpropagation

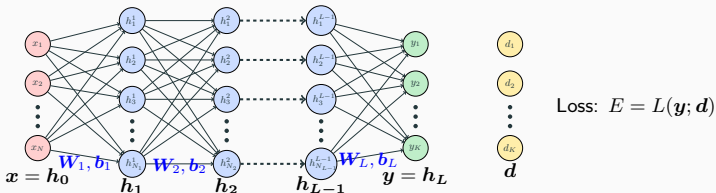


## Gradients for last layer parameters

Given the gradient with respect to the output layer  $\nabla_{h_L} E$ , so far we can compute:

- $\nabla_{a_L} E = \nabla_{h_L} E \odot g'_L(a_L)$
- $\nabla_{b_L} E = \nabla_{a_L} E$
- $\nabla_{W_L} E = \nabla_{a_L} E h_{L-1}^T$

# ANN – Backpropagation



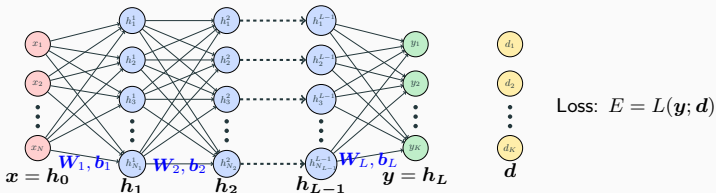
## Gradients for last layer parameters

Given the gradient with respect to the output layer  $\nabla_{\mathbf{h}_L} E$ , so far we can compute:

- $\nabla_{\mathbf{a}_L} E = \nabla_{\mathbf{h}_L} E \odot g'_L(\mathbf{a}_L)$
- $\nabla_{\mathbf{b}_L} E = \nabla_{\mathbf{a}_L} E$
- $\nabla_{\mathbf{W}_L} E = \nabla_{\mathbf{a}_L} E \mathbf{h}_{L-1}^T$

How can we compute the gradients for the parameters of layer  $L - 1$ ?

# ANN – Backpropagation



## Gradients for last layer parameters

Given the gradient with respect to the output layer  $\nabla_{\mathbf{h}_L} E$ , so far we can compute:

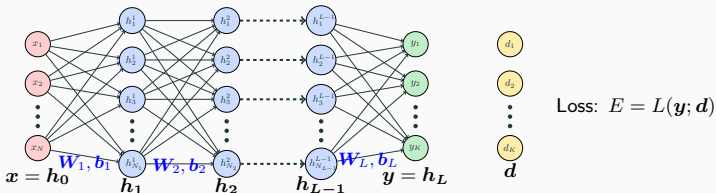
- $\nabla_{\mathbf{a}_L} E = \nabla_{\mathbf{h}_L} E \odot g'_L(\mathbf{a}_L)$
- $\nabla_{\mathbf{b}_L} E = \nabla_{\mathbf{a}_L} E$
- $\nabla_{\mathbf{W}_L} E = \nabla_{\mathbf{a}_L} E \mathbf{h}_{L-1}^T$

## How can we compute the gradients for the parameters of layer $L - 1$ ?

We need the expression of the gradient with respect to the last but one hidden layer  $\mathbf{h}_{L-1}$ ... and then the same formulas apply!

$$\nabla_{\mathbf{h}_{L-1}} E = ?$$

# ANN – Backpropagation

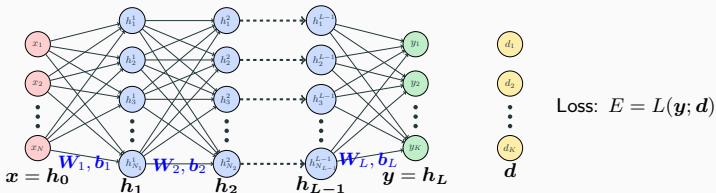


## Gradient with respect to the last but one hidden layer $h_{L-1}$

Here, even to compute the scalar partial derivative  $\frac{\partial E}{\partial h_j^{L-1}}$ , we need to use differential calculus for multivariate functions since  $h_j^{L-1}$  appears in each component of  $a_L$ :

$$\text{For all } i \in \{1, \dots, N_L\}, a_i^L = \sum_{j=1}^{N_{L-1}} w_{i,j}^L h_j^{L-1} + b_i^L.$$

# ANN – Backpropagation



## Gradient with respect to the last but one hidden layer $h_{L-1}$

Let us recall the derivative rule for composition with affine maps:

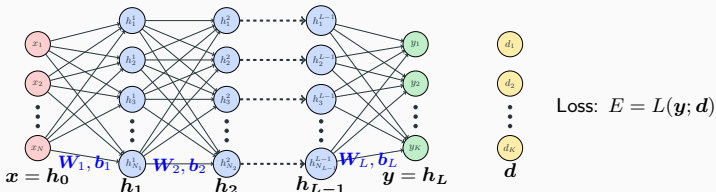
$$\text{For } \varphi(x) = f(Ax + b) \text{ one has } \nabla \varphi(x) = A^T \nabla f(Ax + b).$$

Using the decomposition

$$\begin{aligned} \mathbb{R}^{N_{L-1}} &\rightarrow \mathbb{R}^{N_L} \rightarrow \mathbb{R} \\ h_{L-1} &\mapsto a_L = W_L h_{L-1} + b_L \mapsto E \end{aligned}$$

$$\text{Vector formula: } \nabla_{h_{L-1}} E = W_L^T \nabla_{a_L} E$$

# ANN – Backpropagation



## Forward pass

Initialization:

$$h_0 = x$$

**for** layer  $k = 1$  **to**  $L$  **do**

Linear unit:

$$a_k = W_k h_{k-1} + b_k$$

Componentwise non-linear activation:

$$h_k = g_k(a_k)$$

**end**

Output layer:

$$y = h_L$$

Compute loss:

$$E = L(y; d)$$

## Backward pass

Initialization: Gradient of output layer:

$$\nabla_{h_L} E = \nabla L(y; d)$$

**for** layer  $k = L$  **to** 1 **do**

Componentwise gain of error:

$$\delta_k = \nabla_{a_k} E = \nabla_{h_k} E \odot g'_k(a_k)$$

Gradient of layer bias:

$$\nabla_{b_k} E = \delta_k$$

Gradient of weights:

$$\nabla_{W_k} E = \delta_k h_{k-1}^T$$

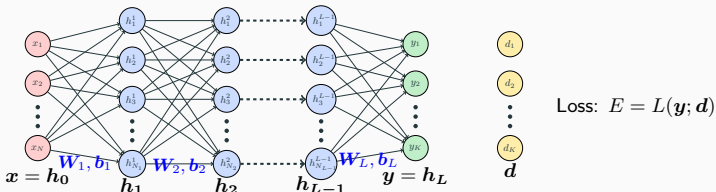
Gradient of previous hidden layer:

$$\nabla_{h_{k-1}} E = W_k^T \delta_k$$

**end**



# ANN – Backpropagation



## Forward pass

Initialization:

$$\mathbf{h}_0 = \mathbf{x}$$

**for** layer  $k = 1$  **to**  $L$  **do**

Linear unit:

$$\mathbf{a}_k = \mathbf{W}_k \mathbf{h}_{k-1} + \mathbf{b}_k \text{ (stored)}$$

Componentwise non-linear activation:

$$\mathbf{h}_k = g_k(\mathbf{a}_k) \text{ (stored)}$$

**end**

Output layer:

$$\mathbf{y} = \mathbf{h}_L$$

Compute loss:

$$E = L(\mathbf{y}; \mathbf{d})$$

## Backward pass

Initialization: Gradient of output layer:

$$\nabla_{\mathbf{h}_L} E = \nabla L(\mathbf{y}; \mathbf{d})$$

**for** layer  $k = L$  **to** 1 **do**

Componentwise gain of error:

$$\delta_k = \nabla_{\mathbf{a}_k} E = \nabla_{\mathbf{h}_k} E \odot g'_k(\mathbf{a}_k)$$

Gradient of layer bias:

$$\nabla_{\mathbf{b}_k} E = \delta_k$$

Gradient of weights:

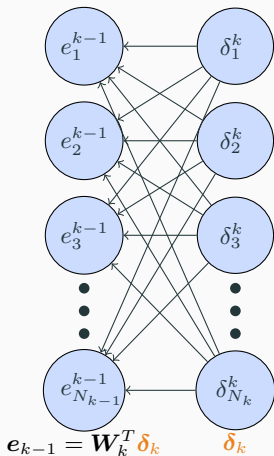
$$\nabla_{\mathbf{W}_k} E = \delta_k \mathbf{h}_{k-1}^T$$

Gradient of previous hidden layer:

$$\nabla_{\mathbf{h}_{k-1}} E = \mathbf{W}_k^T \delta_k$$

**end**

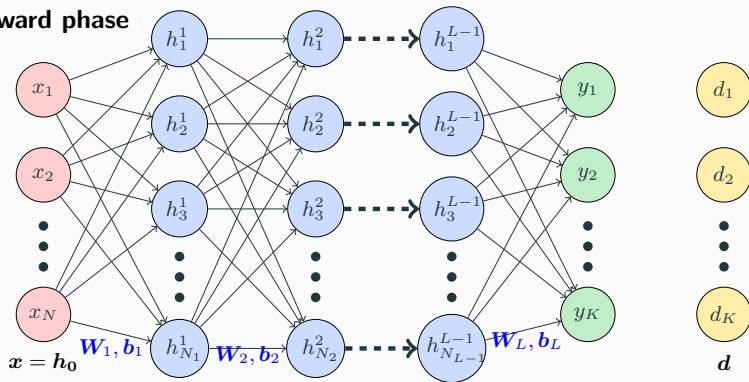
## Error backpropagation



- Gradient of previous hidden layer:  
$$e_{k-1} = \nabla_{h_{k-1}} E = \mathbf{W}_k^T \boldsymbol{\delta}_k$$
- Multiplying by  $\mathbf{W}_k^T$  corresponds to passing to the linear layer in reverse order.
- The error is backpropagated layer by layer to compute the gradient with respect to each layer parameters.

## Error backpropagation

Forward phase



Input Layer

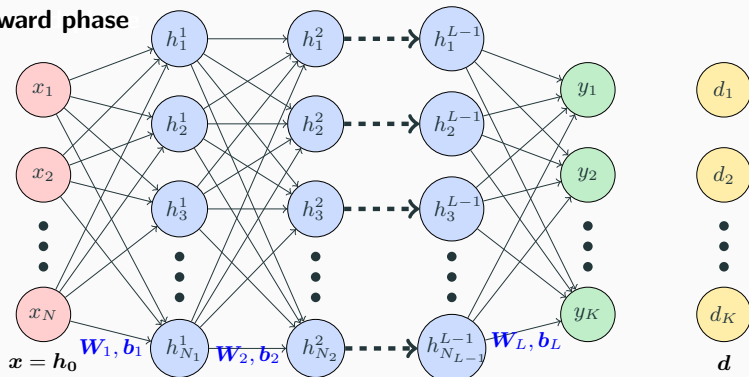
Hidden Layers

Output Layer

Label

## Error backpropagation

Forward phase



$$a_1 = W_1 h_0 + b_1$$

$$h_1 = g_1(a_1)$$

$$l = \sum_k d_k y_k$$

Input Layer

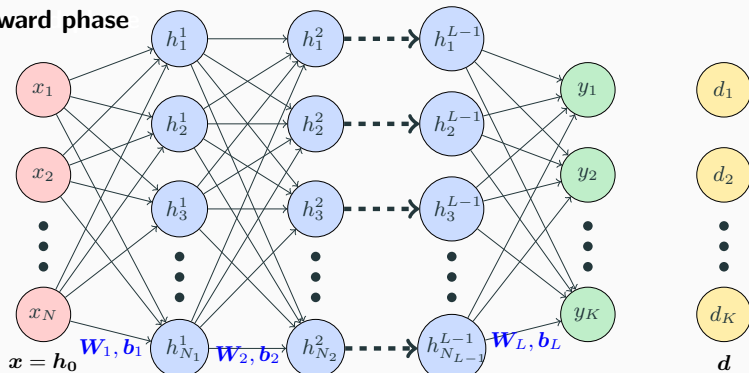
Hidden Layers

Output Layer

Label

## Error backpropagation

Forward phase



Input Layer

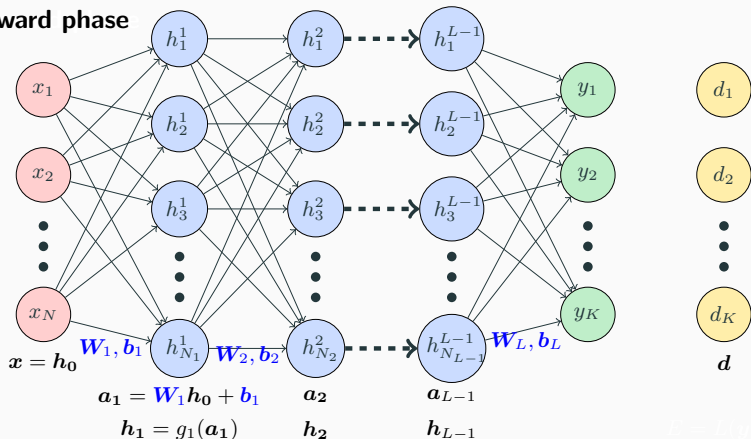
Hidden Layers

Output Layer

Label

## Error backpropagation

Forward phase



Input Layer

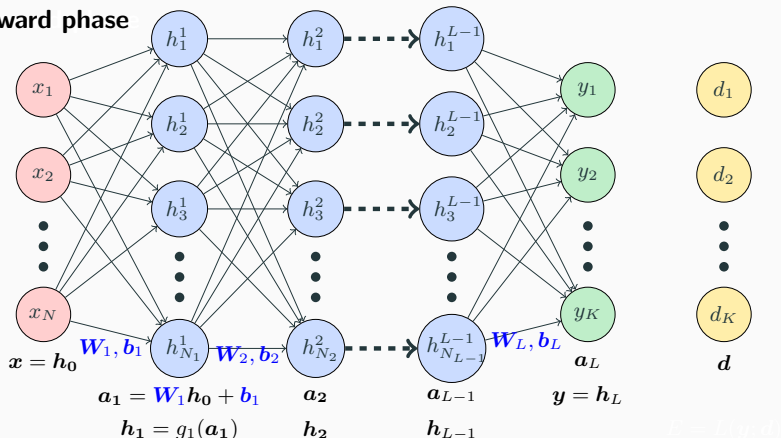
Hidden Layers

Output Layer

Label

## Error backpropagation

Forward phase



Input Layer

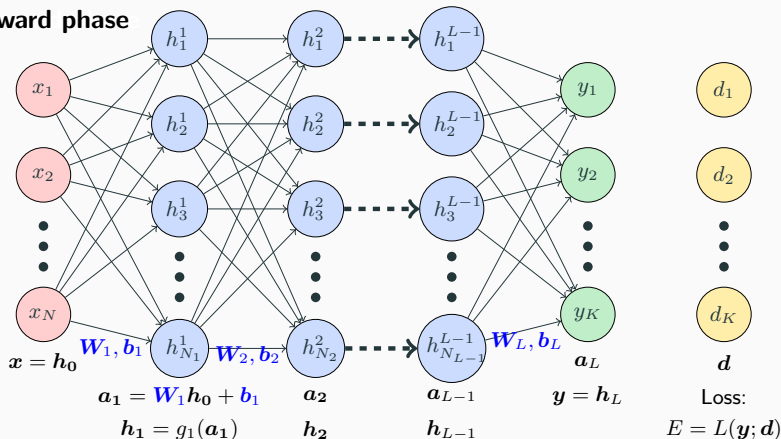
Hidden Layers

Output Layer

Label

## Error backpropagation

Forward phase



Input Layer

Hidden Layers

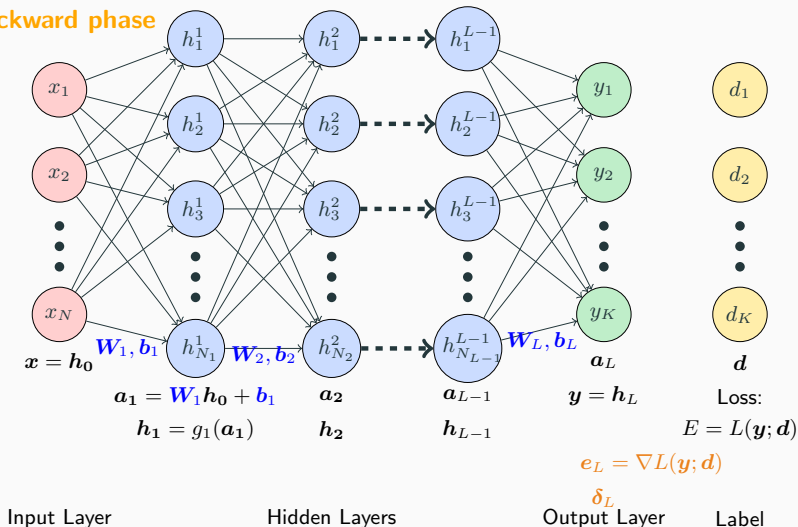
Output Layer

Label



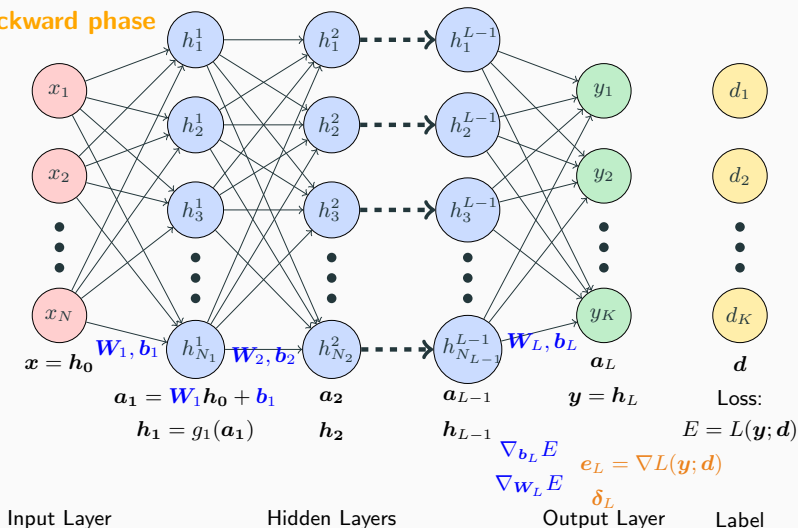
## Error backpropagation

### Backward phase



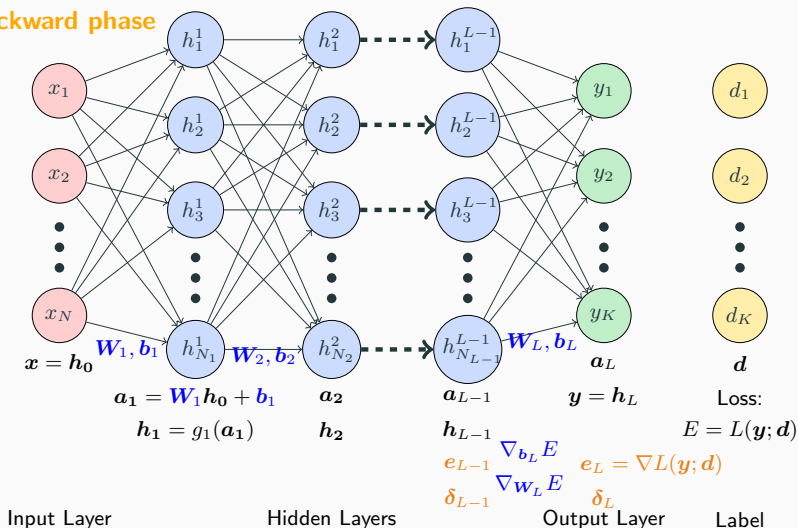
## Error backpropagation

### Backward phase



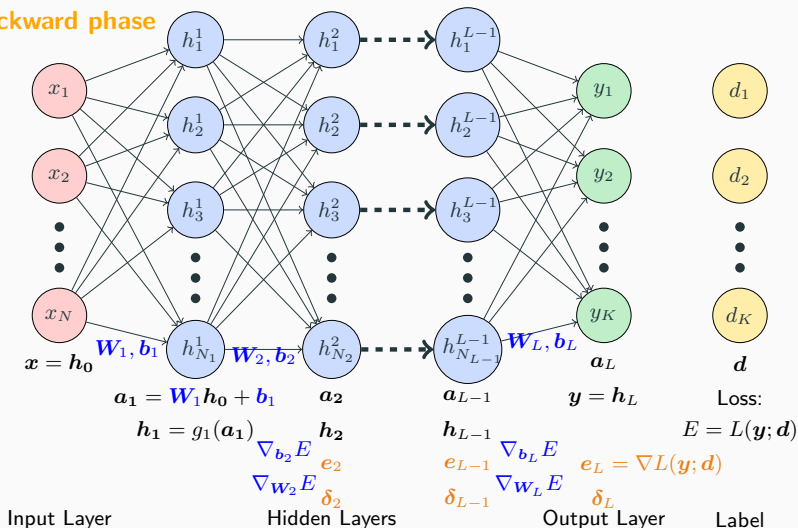
## Error backpropagation

### Backward phase



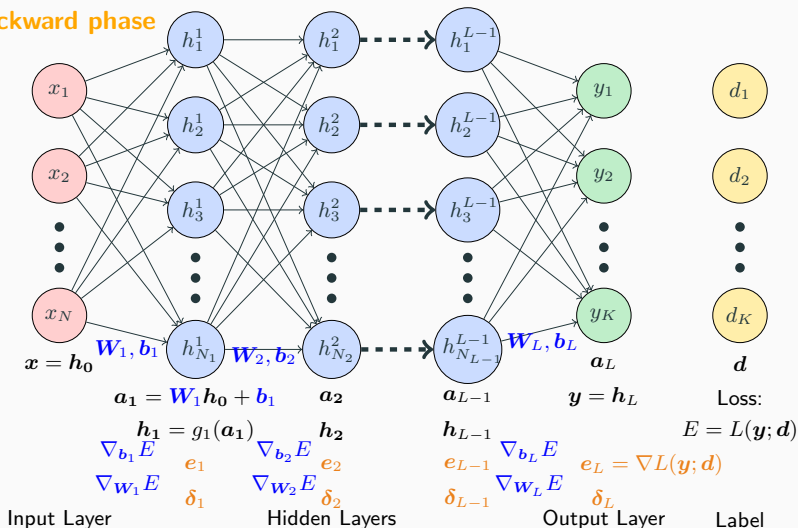
## Error backpropagation

### Backward phase



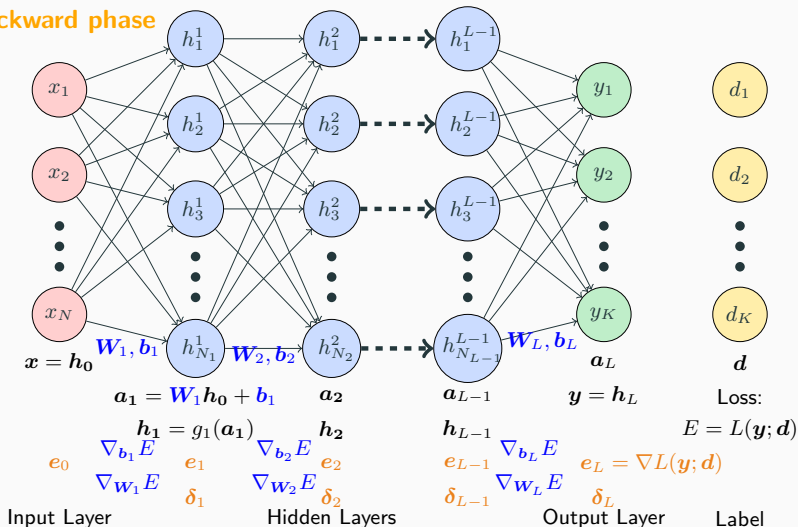
## Error backpropagation

### Backward phase



## Error backpropagation

### Backward phase



## Error backpropagation in practice

Training loss:

$$E(\mathbf{W}) = \sum_{(\mathbf{x}^i, \mathbf{d}^i) \in \mathcal{T}} L(\mathbf{y}^i; \mathbf{d}^i)$$

- The backpropagation procedure computes  $\nabla_{\mathbf{W}} L(\mathbf{y}^i; \mathbf{d}^i) = \nabla_{\mathbf{W}} L(f(\mathbf{x}^i; \mathbf{W}); \mathbf{d}^i)$ .
- This has to be done for each data point  $\mathbf{x}^i \in \mathcal{T}$ .
- By linearity, the final gradient  $\nabla E(\mathbf{W})$  is the sum of all individual gradients  $\nabla_{\mathbf{W}} L(\mathbf{y}^i; \mathbf{d}^i)$ .
- These gradients are summed sequentially (no need to store each individual gradients).
- In general we do not compute the exact gradient...

## Error backpropagation in practice

Batch loss:

$$E(\mathbf{W}) \approx \sum_{(\mathbf{x}^i, \mathbf{d}^i) \in \mathcal{S}} L(\mathbf{y}^i; \mathbf{d}^i), \quad \text{with } \mathcal{S} \subset \mathcal{T}$$

- The backpropagation has to be done for each visited data point  $\mathbf{x}^i \in \mathcal{S}$  of the batch.
- The gradient for each point  $\mathbf{x}^i$  is added to the running gradient = current gradient estimation.
- Once the noisy estimated gradient is used as a gradient step, one needs to set the gradients to zero: See PyTorch `torch.zero_grad()` procedure.



# Questions?

---

Sources, images courtesy and acknowledgment

Charles Deledalle

V. Lepetit

L. Masuch